From Theory to Practice: Sebesta Chapter 5 and 9 Meets PLAI

A Hands-On Workshop Bridging Concepts with Implementation

This is an ambitious endeavor that will culminate in implementing your own programming language.

Let’s play with SMoL and Stacker

Using def1, def2, def2, and post1 and trace using stacker

The Stacker experience

Here is Stacker: https://smol-tutor.xyz/stacker/

Copy/paste and trace the code in Stacker

```
#lang stacker/smol/hof

;; Sebesta 5.4.3: Stack-dynamic variables
;; Let's SEE them in action!

(deffun (f x)
  (+ x 1))

(deffun (g y)
  (* y 2))

(g (f 5))

;; Step through this in the Stacker
;; Watch the stack grow and shrink!
```

1. Load this in Stacker
2. Step through execution
3. Identify:
    - When variables are allocated (Sebesta: elaboration time)
    - When they're deallocated
    - The environment at each step

More Complex Example:

```
#lang stacker/smol/hof

;; Sebesta's static vs stack-dynamic distinction

(defvar global-x 100)  ;; This is STATIC (allocated before execution)

(deffun (make-adder n)
  (defvar local-y n)   ;; This is STACK-DYNAMIC (allocated on call)
  (+ global-x local-y))

(make-adder 5)
(make-adder 10)

;; Run this in Stacker
;; Watch how local-y is created and destroyed
;; But global-x stays the same
```

Questions:

1. When is `local-y` allocated? Deallocated?
2. Sebesta says stack-dynamic variables can't be history-sensitive (page 222). What does the Stacker show about this?
3. How many DIFFERENT `local-y` variables exist during this program?

Let’s install Racket and play with DrRacket

https://racket-lang.org/

Let’s start reading page 16 and play with the code, copy/pasted in Racket.

Activity 1: The Binding Detective

Duration: 45 minutes
Sebesta Connection: 5.4 (Binding and Binding Times)
PLAI Pages: 46-56

Part A: Binding Time Investigation

Setup: Open DrRacket and create a new file with `#lang plait`

```
#lang plait
```

```
;; Sebesta talks about different binding times.
;; Let's see them in action!

;; 1. DESIGN TIME binding
;; The + operator is bound to addition at language design time
(define result1 (+ 2 3))

;; 2. IMPLEMENTATION TIME binding
;; Number range is decided when Racket was implemented
(define big-number 999999999999999999)

;; 3. COMPILE TIME binding (in plait, this is type checking time)
;; This will fail at compile time - uncomment to see:
;; (define oops (+ "hello" 5))

;; 4. RUN TIME binding
;; The actual value is bound when we run
(define (add-numbers x y)
  (+ x y))

(test (add-numbers 10 20) 30)
```

Questions

1. What happens when you uncomment the `oops` line? When is this error caught?
2. In Sebesta's terms, what is the binding time for the type of `result1`?
3. If you change `big-number` to a bigger value, when do you find out if it's too big?

Part B: Static vs Dynamic Binding

Create this example (adapting PLAI page 54):

```
#lang plait

;; STATIC BINDING (what plait uses)
;; Type is bound at compile time, never changes

(define x 5)              ;; x is bound to Number type
;; (define x "hello")     ;; Can't rebind to different type!

;; Let's see how local bindings work (Sebesta 5.5.1)

(define-type Exp
  [numE (n : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE (name : Symbol)]
  [let1E (var : Symbol) (value : Exp) (body : Exp)])

;; Now write a mini interpreter!
```

```
;; This shows HOW bindings actually work

(define-type-alias Env (Hashof Symbol Number))
(define mt-env (hash empty))

(define (lookup (s : Symbol) (env : Env)) : Number
  (type-case (Optionof Number) (hash-ref env s)
    [(none) (error s "not bound")]
    [(some v) v]))

(define (extend (env : Env) (name : Symbol) (val : Number)) : Env
  (hash-set env name val))

(define (interp (e : Exp) (env : Env)) : Number
  (type-case Exp e
    [(numE n) n]
    [(varE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(let1E var val body)
     (let ([new-env (extend env var (interp val env))])
       (interp body new-env))]))

;; Test it! (Sebesta's example from page 220)
;; In C-like syntax this would be:
;; {
;;   int x = 5;
;;   x + x
;; }

(test (interp (let1E 'x (numE 5)
                     (plusE (varE 'x) (varE 'x)))
              mt-env)
      10)

;; Nested bindings (Sebesta page 220)
(test (interp (let1E 'x (numE 3)
                     (let1E 'y (numE 4)
                            (plusE (varE 'x) (varE 'y))))
              mt-env)
      7)
```

Discussion Questions:

1. The `Env` type represents what Sebesta calls a "referencing environment" (5.7). How does it work?
2. In `extend`, we create a NEW environment. Why not modify the old one?
3. Compare this to Sebesta's description on page 218. What's the same? Different?

Activity 2: Scope Detective Work

Sebesta Connection: 5.5 (Scope - Static vs Dynamic)
PLAI Pages: 51-53, 65-67

Part A: The Classic Scope Problem (20 min)

Present this scenario (from PLAI page 51-52):

```
#lang plait

;; Sebesta 5.5.1: Static Scope
;; PLAI page 51: "Static scope is a defining characteristic of SMoL"

;; Here's the problem: What should this print?

;; In pseudocode:
;; x = 1
;; function f(y):
;;   return x + y
;;
;; function g():
;;   x = 2
;;   return f(3)
;;
;; print(g())

;; Question: Does it print 4 or 5?

;; Let's implement both scoping rules to see!

;; First, let's extend our Exp type:
(define-type Exp
  [numE (n : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE (name : Symbol)]
  [let1E (var : Symbol) (value : Exp) (body : Exp)]
  [lamE (var : Symbol) (body : Exp)]
  [appE (fun : Exp) (arg : Exp)])

(define-type Value
  [numV (n : Number)]
  [closV (var : Symbol) (body : Exp) (env : Env)])

(define-type-alias Env (Hashof Symbol Value))
(define mt-env (hash empty))

;; STATIC SCOPE interpreter (the RIGHT way - SMoL way)
(define (interp-static (e : Exp) (env : Env)) : Value
  (type-case Exp e
    [(numE n) (numV n)]
```

```
    [(varE s)
     (type-case (Optionof Value) (hash-ref env s)
       [(none) (error s "not bound")]
       [(some v) v])]
    [(plusE l r)
     (let ([lv (interp-static l env)]
           [rv (interp-static r env)])
       (numV (+ (numV-n lv) (numV-n rv))))]
    [(lamE var body)
     ;; KEY: Save the environment NOW! (Static scope)
     (closV var body env)]
    [(appE fun arg)
     (let ([funV (interp-static fun env)]
           [argV (interp-static arg env)])
       (type-case Value funV
         [(closV var body saved-env)
          ;; Use the SAVED environment, not current!
          (interp-static body (hash-set saved-env var argV))]
         [else (error 'app "not a function")]))]
    [(let1E var val body)
     (let ([new-env (hash-set env var (interp-static val env))])
       (interp-static body new-env))]))

;; The example in our language:
;; x = 1
(define prog1
  (let1E 'x (numE 1)
    ;; f = λy. x + y
    (let1E 'f (lamE 'y (plusE (varE 'x) (varE 'y)))
      ;; g = λ(). let x=2 in f(3)
      (let1E 'g (lamE 'dummy
                      (let1E 'x (numE 2)
                             (appE (varE 'f) (numE 3))))
        ;; g()
        (appE (varE 'g) (numE 0))))))

(test (interp-static prog1 mt-env) (numV 4))  ;; Static scope: 1 + 3 = 4
```

Class Discussion:

1. Run this code. What does it print?
2. Why? Where is `x` bound when `f` looks it up?
3. Draw the environment chain (refer to Sebesta Figure 5.3 if available)

Part B: The Moon Example (PLAI page 51-52) (20 min)

Type this together as a class:

```
#lang racket

;; PLAI page 51-52: The most convincing argument for static scope

;; Imagine this pseudocode:
```

```
;; x = 1
;; if moon-is-full():
;;     x = 2
;; print(x)

;; What should this print?
;; With dynamic scope: depends on the moon!

;; Let's simulate this:

(define (moon-is-full)
  ;; Simulate: returns #t or #f randomly
  (< (random 10) 5))

;; Version 1: Static scope (good!)
(define x-static 1)
(define result-static x-static)  ;; Always 1, moon doesn't matter

;; Version 2: Simulating dynamic scope (bad!)
(define x-dynamic 1)
(when (moon-is-full)
  (set! x-dynamic 2))

(printf "Static scope: ~a (always the same)~n" result-static)
(printf "Dynamic scope simulation: ~a (changes randomly!)~n" x-dynamic)
(printf "~nRun this several times - see the difference?~n")
```

Questions:

1. Run this 5 times. What happens?
2. Sebesta says static scope makes programs more reliable (page 217). Why?
3. PLAI says "Dynamic scope is one of the unambiguously wrong design decisions" (page 52). Too harsh?

Part C: Hidden Variables

Sebesta page 220, PLAI page 52:

```
#lang plait

;; Sebesta discusses "hiding" on page 220
;; What happens when the same name is used in nested scopes?

(define prog-hidden
  (let1E 'x (numE 10)
    (let1E 'x (numE 20)
      (varE 'x))))

;; What should this evaluate to?
(test (interp-static prog-hidden mt-env) (numV 20))
```

```
;; The outer x is "hidden"

;; Now a trickier one:
(define prog-unhidden
  (let1E 'x (numE 10)
    (let1E 'y (numE 20)
      (plusE (varE 'x) (varE 'y)))))

(test (interp-static prog-unhidden mt-env) (numV 30))
;; Here x is NOT hidden, because y doesn't hide it

;; Connect to Sebesta's block example (page 220):
;; void sub() {
;;   int count;
;;   while (...) {
;;     int count;  // This hides the outer count
;;     count++;
;;   }
;; }
```

Activity 4: Type Binding Laboratory

Duration: 60 minutes
Sebesta Connection: 5.4.2 (Type Bindings)
PLAI Pages: 109-114, 172-177

Part A: Static Type Binding

Comparison Experiment:

```
#lang plait

;; Sebesta 5.4.2.1: Static Type Binding
;; In plait (like Java, C++), types are bound at compile time

(define x : Number 5)
;; (define x : Number "hello")  ;; Compile error!

;; This is EXPLICIT declaration (Sebesta page 197)

;; But plait also has type INFERENCE:
(define y 10)  ;; Type is inferred to be Number

;; Test the difference:
(+ x y)  ;; Works fine

;; Sebesta page 199: "implicit declarations can be detrimental to
reliability"
;; Let's see if plait's type inference has this problem:
```

```
;; (define typo1 10)
;; (+ typo2 5)  ;; Will this catch the typo? Try it!
```

Now compare with Racket (dynamic typing):

```
#lang racket

;; Sebesta 5.4.2.2: Dynamic Type Binding
;; In Racket (like Python, JavaScript), types are bound at runtime

(define x 5)
(define x "hello")  ;; No error! x changes type
(define x '(1 2 3)) ;; Changes type again!

;; Sebesta's example from page 200:
(define i 10)
(define y #(1 2 3))  ;; An array/vector

;; In a statically typed language, this would be caught:
;; (define i y)  ;; Type error!

;; But in Racket:
(define i y)  ;; No error, i is now an array
(printf "i is now: ~a~n" i)
```

Discussion:

1. Which caught your typo? When?
2. Sebesta lists advantages and disadvantages (pages 199-201). Which do you find most convincing?
3. Run the Racket version. Does it behave as Sebesta predicts?

Part B: Gradual Typing

This is cutting-edge! (PLAI pages 172-177)

```
#lang typed/racket

;; Gradual typing: Mix static and dynamic!

(: add-numbers (-> Number Number Number))
(define (add-numbers x y)
  (+ x y))

;; Static type checking for add-numbers

;; But we can also have dynamic:
(define mystery-value 42)  ;; Type inferred

;; And even ANY type:
(: flexible-function (-> Any Number))
(define (flexible-function x)
```

```
  (if (number? x)
      (+ x 1)
      0))

(test (flexible-function 5) 6)
(test (flexible-function "hello") 0)
```

Challenge: Sebesta doesn't cover gradual typing. Research it and answer:

1. Why would you want to mix static and dynamic typing?
2. What are the trade-offs?
3. Find a real-world project that uses TypeScript or Typed Racket. Why did they choose gradual typing?

Activity 5: Build Your Own Mini-Language (Mini project – Team group)

Integration: All of Chapter 5 + PLAI approach

The Challenge

Create a small language that demonstrates YOUR understanding of:

- Binding times
- Scoping rules
- Type binding
- Variable lifetime

Starter Template

```
#lang plait

;; YOUR LANGUAGE: Design decisions to make!

;; 1. What data types will you support?
(define-type Exp
  [numE (n : Number)]
  [boolE (b : Boolean)]
  ;; TODO: Add more!
  )

;; 2. What scoping rule? (You choose!)
;; - Static scope (like we've been doing)
;; - Something creative?
```

```
;; 3. Will you have:
;; - Only static variables?
;; - Stack-dynamic variables?
;; - Functions? (If yes, closures!)

(define-type Value
  [numV (n : Number)]
  [boolV (b : Boolean)]
  ;; TODO: Add more!
  )

;; 4. Write your interpreter!
(define (my-interp (e : Exp) (env : Env)) : Value
  ;; TODO: Implement based on your design decisions!
  (numV 0))  ;; Placeholder

;; 5. Write tests that demonstrate:
;;    - Binding happening correctly
;;    - Scoping working as you designed
;;    - Type binding (if applicable)

;; Example test structure:
;; (test (my-interp YOUR-PROGRAM mt-env) EXPECTED-RESULT)
```

Requirements

Your language must demonstrate:

1. **At least 2 binding times** from Sebesta 5.4
2. **Clear scoping rule** (document it!)
3. **At least 3 test cases** showing different scenarios
4. **Written explanation** connecting to Sebesta concepts

Presentation

Each student (or group) presents:

1. Design decisions (3 min)
2. Live demo (2 min)
3. One "cool thing" your language can do (1 min)

Extension (Bonus)

Advanced: Implement Dynamic Scope

```
#lang plait
```

```
;; Challenge: Modify interp-static to use DYNAMIC scope
;; Hint: Don't save the environment in the closure!

(define (interp-dynamic (e : Exp) (env : Env)) : Value
  (type-case Exp e
    ;; ... most cases same ...
    [(lamE var body)
     ;; TODO: What should you save here?
     (error 'TODO "Implement dynamic scope!")]
    [(appE fun arg)
     ;; TODO: What environment should you use?
     (error 'TODO "Implement dynamic scope!")]))

;; Test with prog1 from Activity 2
;; Should now get 5 instead of 4!
```

Research Project: Real-World Language Comparison

Pick 2 languages with different design choices from Chapter 5:

- One with static typing, one with dynamic (e.g., Java vs Python)
- Both with static scope (most modern languages)

Write a 2-page report:

1. How do their binding time decisions affect programming?
2. Find a real bug that could only happen in one language
3. Which do you prefer for what tasks?

Language Design Debate

- Team Static: Defend static typing and scoping
- Team Dynamic: Defend dynamic typing and scoping
- Team Gradual: Defend mixing both

Format:

- 3-minute opening statement
- 5-minute rebuttal
- Audience questions
- Vote on most convincing